The EAST-ADL: A Joint Effort of the European Automotive Industry to Structure Distributed Automotive Embedded Control Software
Ulrich Freund, Henrik Lönn, Evelyne Silva, Jörn Migge, Denis Bugnot, Bruno Godard, Matthias Weber, Mark-Oliver Reiser

To cite this version:
Ulrich Freund, Henrik Lönn, Evelyne Silva, Jörn Migge, Denis Bugnot, et al.. The EAST-ADL: A Joint Effort of the European Automotive Industry to Structure Distributed Automotive Embedded Control Software. 2nd Embedded Real Time Software Congress (ERTS’04), 2004, Toulouse, France. hal-02275442

HAL Id: hal-02275442
https://hal.science/hal-02275442
Submitted on 30 Aug 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Session 7A: Major Projects

The EAST-ADL: A Joint Effort of the European Automotive Industry to Structure Distributed Automotive Embedded Control Software

Ulrich Freund, ETAS GmbH, Stuttgart
Henrik Lönn, Volvo Technology Corporation, Gothenburg
Evelyne Silva, Jörn Migge, PSA Peugeot-Citroën, Vélizy
Denis Bugnot, Valeo, Créteil
Bruno Godard, Siemens VDO, Toulouse
Matthias Weber, Mark-Oliver Reiser DaimlerChrysler, Berlin

Introduction

Around ninety percent of vehicle innovations are driven mainly by electronics. Hence, software- and systems-engineering becomes a crucial discipline which vehicle manufacturers and their suppliers have to conquer. Automotive software runs on so-called Electronic Control Units (ECU). Besides a microcontroller and memory an ECU consists of power electronics to drive sensors and actuators. The software implementing control algorithms combines the sensor values and calculates some meaningful actuator signals. ECU software- and system-engineering is characterized by the following attitudes:

- It is embedded software which means it directly interacts with sensors and actuators and does not change its purpose during lifetime.
- The software fulfills a dedicated control task, i.e. the performance of the control algorithm imposes real-time constraints on the software to be fulfilled.
- The software is realized as a distributed system. Since the information of several sensors provided can (and hence will) be used to improve the control algorithm's performance the information has to be sent to several other ECUs.
- The development itself is distributed. As a rule, a vehicle manufacturer employs several suppliers to deliver the control algorithms and the ECU. Since both the algorithms as well as the ECUs have to work together the vehicle manufacturer has to do a lot of integration work to get the vehicle on the road.

Since vehicle manufactures traditionally put their focus on production cost rather than on development cost, the sensors and actuators along with the bare ECU represent almost the whole amount of costs for electronics spent. Though software does not have a direct amount of production cost it is not for free! The only parameter where software contributes directly to production cost is by memory size. It is therefore a must for a software to be as tiny as possible. Furthermore, there is a direct relationship between the amount of production cost for sensors/actuators and the complexity of the control software in between.
The EAST-EEA project was formed as an ITEA project by working groups from EUCAR & CLEPA in mid 2001 and runs to mid 2004. 23 partners from vehicle manufacturers, suppliers, tool-suppliers and research institutes contribute to define a common middleware as well as an architecture description language. The results are validated by the vehicle manufacturers and suppliers.

This paper starts with a bottom-up consideration how embedded basic software components are identified to structure embedded software programs. Afterwards the EAST-ADL is introduced from a vehicle manufacturer’s point of view. The middleware is explained on the body electronics domain while the following section shows how ADL and middleware are incorporated in the design of distributed embedded automotive control software.

Embedded Basic Software

To meet the constraints of series production ECUs several programming techniques have been identified in the last twenty years:
- Establishing building blocks and sub blocks. Exchange of information between building blocks asynchronously by means of messages (ECU global variables) instead of synchronous procedure calls.
- Building blocks are implemented by using so-called “void-void functions”, i.e. subroutines having neither formal parameters nor return values.
- Call stack minimization due to limited synchronous function calls within a building block.
- Use of message duplication in case of tentative task interruption. Cyclic tasks with a well chosen cycle time ensure the meeting of hard real-time constraints.
- To save overall vehicle’s weight and wiring harness, bus-systems are preferred for the communication between ECUs.
- In case of bus-interconnected ECUs, cyclic broadcasting mechanisms with collision resolution (e.g. CAN) are the preferred communication means.
- Explicit and static scheduling of functions within building blocks.

At a first glance these techniques differ considerably from the layered approaches proposed in computer-science. However, taking a deeper look into embedded automotive software, one can identify building blocks for basic functions like communication drivers or peripheral access routines as well as building blocks for control-algorithms. Last but not least, there are program fragments realizing a scheduler and global variables for data-exchange between the void-void functions. These building blocks can be layered as depicted in Figure 1.
In this layered interpretation of the building blocks the middleware takes a prominent role because it encapsulates the global variables for data-exchange. Access in the void-void functions to the middleware will be realized by an API instead of a direct access to the global variables. This API might be realized by macros.

The program fragments realizing the OS-scheduler as well as the building blocks for the communication layer or the hardware-abstraction layer are (or will be soon) standardized by OSEK-VDX\(^1\) and HIS\(^2\). The feasibility of a middleware has been shown independently by the French AEE project introducing the ICEM[4] (Inter-Component-Exchange-Manager) or by DaimlerChrysler's ORPC[3] (OSEK-based Remote Procedure Call). The latter has been prototypically realized in the TITUS-Tool-Suite. Both projects have shown that introducing a middleware has a considerable impact on the structure of the building blocks for application functions. To design these building blocks properly, an Architecture Description Language (ADL) is indispensable.

**The EAST-ADL**

Managing complexity in automotive electronics is the approach to transform vehicle level functional and non-functional requirements to structured distributed automotive embedded control software running on an appropriate electronic architecture. The EAST-ADL is introduced to provide means to facilitate the numerous approaches existing currently in the European automotive industry. To cope with the complexity the EAST-ADL is structured into several abstraction levels as shown in Figure 2. Wherever possible, the EAST-ADL is based on UML 2.0 concepts.

---

1 OSEK-VDX is a standard for embedded automotive operating systems
2 HIS (Hersteller Initiative Software) is an initiative of German vehicle manufacturers to standardize embedded basic software components
The goal of the *vehicle project* (VP) is to describe the product line architecture for a vehicle. An Electronic Feature is a user-visible aspect of a system that is implemented by the electronic architecture of the system. Electronic features may be linked by the "is a variant of"-relationship. The goal is to define a structured view of the inherent variability. For example, the "Wiper System with Rain sensor" is a variant of "Advanced Wiper System" which is a variant of "Wiper System". The vehicle type concept is introduced to allow the hierarchical structuring of vehicle architecture variants. For each vehicle type a variant is defined by tagging the appropriate electronic features from the library. Furthermore, a vehicle type may inherit the mandatory or optional electronic features of another, more general, vehicle type. This is expressed by an "inheritance"-relation (generalisation, to be more precisely). Additionally, an optional electronic feature of a more general vehicle type can be specified as mandatory for a more specific vehicle type but not vice versa. Last but not least, it is possible to exclude an "inherited" electronic feature in a vehicle type explicitly.

The *functional analysis architecture* (FAA) is a description of the electronic architecture from a functional point of view. In other words, it is a precise answer to the question "What does the system do?", but independently of software implementation concerns ("How does the system do it?"). This implies that the way the analysis functions are structured is likely to be closer to the structuring of the user visible electronic features than the structuring of software functions implementing later on these features. The goal is to have a precise technical specification of the electronic architecture that allows checking desired properties at system level. The functional analysis architecture could for example support a simulation of the global behavior of the control algorithms. For this purpose, the functional analysis architecture consists of hierarchically decomposed analysis functions that may communicate with each other through signals. In parallel to the hierarchical decomposition, analysis functions may be linked by "is a variant of"-relations. If an analysis function is a variant of another analysis function the latter

---

**Figure 2: EAST-ADL Abstraction Layers**
resolves "variation point" of the former. If an analysis function having variants is used somewhere in the decomposition hierarchy it means that at this "point" any of its variants may be inserted. Hence, the interface of an Analysis Function with variants is the union of the interfaces of its variants. The combination of the decomposition hierarchy and the variant hierarchy allows to "factor out" what is common and thus obtain a compact description of the variability inherent in the functional analysis architecture due to the vehicle type hierarchy. Analysis functions contribute to the realization of electronic features. These links are expressed through a special relation called "selects". The terminology stems from variant modeling. The selects relation enables traceability of dependencies.

Figure 3: Composite Software Function

The functional design architecture (FDA) is the most abstract description of software structures to be found later in the ECU software implementation. This architecture consists of a functional hierarchy with focus on typed structures and signal exchange. Typed structures are composite software functions and elementary software functions. Elementary software functions are the leaves of a hierarchy whereas composite software functions can contain other composite software functions or elementary software functions as parts. All components have interface types describing the required and provided "services" of the component. Interfaces can be distinguished in

- Signal-Function-Interfaces
- Operational-Function-Interfaces

Operational-Function Interfaces are method calls in the classical sense of UML 2.0. Signal-Function-Interfaces are used if the application algorithm, e.g. a closed control loop, is data-flow oriented. Data exchange is specified via ports. The port type is determined by the type of their employed interfaces. Ports carry additional information for timing.

In Figure 3 a composite software function for a seat-heat control algorithm is shown. In the center there is an elementary software function for the control algorithm whereas four other composite software functions are used for reading the switch position, indicating the current stage via an LED and driving the seat-heating actuator. This composite software function receives vehicle wide available clamp and
voltage signals. These signals are described by the port symbols including the associated signal function interfaces.

The EAST-ADL employs a two-stage type system, i.e. all variables have an abstract data type as well as an implementation data type. All formal parameters in methods of the Operational-Function-Interfaces or Values of the Signal-Function-Interfaces are at first specified as Abstract-Data-Types. This means that no physical limits or integer arithmetic are considered. In a second step these Abstract-Data-Types will be "refined" to Implementation-Data-Types, now taking into account the specified physical limits of the signal as well as appropriate resolution\[1\]. This results in a number of required bits which have to be “mapped” to an appropriate implementation type, e.g. an 8-bit unsigned integer. Code-generation will later on respect these settings and adjust the operators accordingly. In addition, it is possible to associate behavior with functions of the hierarchy. However, only elementary software functions, can carry behavior. Behavior can be described using finite state machines, difference equations or plain source-code. Separating function types from their tentative instances ensures sparse ROM utilization in later design stages.

The logical architecture (LA) describes the system on pure instance level. Function instances, i.e. instances of elementary software functions, are clustered to so-called Logical-Clusters. Within a Logical-Cluster, function instances communicate synchronously and ensure that tightly coupled function instances are mapped on the same ECU as well as to the same OS-task. While respecting all timing and memory requirements, Logical-Clusters can be mapped freely to ECUs and OS-tasks. In Body Electronic applications, application software components represent logical clusters. ECU properties with their sensors and actuators as well as the bus-systems are described in the hardware architecture (HA). Configurable embedded basic software components for the ECUs like OS, HAL, communication.-software as well as the EAST-middleware establishes the technical architecture (TA). An example from Body electronics is given in the next section.

Last but not least the operational architecture (OA) describes the running system. It is conceived by first mapping the Logical-Clusters of the logical architecture to ECUs and afterwards to OS-tasks running on that ECU. Connectors being mapped to “different ECUs” will be mapped to the bus-message. Code-generation of the application software and configuration of the basic software results in appropriate C-Code for the ECU which has to be compiled, linked, and flashed to the micro-controller of the ECU.

On every level of abstraction except the vehicle project it is possible to attach behavioral descriptions to the model, simulating it or checking the properties of the model and to generate code for an appropriate target, e.g. a PC, an experimental hardware or a series production ECU. Generally speaking, the EAST-ADL is capable of linking design-information horizontally (e.g. simulation and code-generation) and vertically (i.e. between different layers of abstraction).

Behavior is either described by external tools (external behavior) or by native elements. In the first case the behavior being described in an external tool (such as Simulink, Statemate, or ASCET-SD) which can be used for simulation, implementation or analysis. The entity can be attached either to analysis functions in the FAA or to composite software functions and elementary software functions in the FDA. The behavioral description may be associated to a software function with two different aims:

- Specification: this is the case whenever the external behavior will be associated with an analysis software function or a composite software function. It will represent a behavior specification and may be used for simulation and analysis purposes depending on the external tool capabilities.
- Implementation: the external behavior description may represent the chosen behavior implementation only in the association with an elementary software function.

As a rule, behavior modeling tools employ a modeling language consisting of behavioral and structural elements, e.g. blocks & sub-blocks. If such behavioral modeling tools are used to model external behavior, there has to be an equivalence model between the ADL and the structuring elements of the external tool. The structural elements in that tool are derived from the EAST-ADL and thus must not be modified because system integration is again defined in the EAST-ADL. Means for accessing external description parts will be provided in the language for traceability and documentation purposes.

There exists furthermore a native behavior description based on UML 2.0 components. These entities can be used on FDA-level only and are based on object- and interaction-diagrams. A state-chart interpretation with appropriate semantics for series production code generation is currently under definition.

The EAST-Middleware

The EAST-middleware offers services to transmit data transparently between different ECUs and different tasks. The number of bits specified with the implementation-data-types determine the number of bits the communication software has to reserve on the bus-system. In automotive applications, it is usual to map several SDUs\(^3\), i.e. application signals, to one PDU\(^4\), e.g. a CAN-Frame. For communication between logical-clusters within the logical architecture the middleware implements the IPC-exchangers. From the ECU programming point of view, the IPC-exchangers are implemented by ECU-global variables. Access to the IPC-exchangers is provided by an API which code-generation has to respect for the logical-cluster.

\(^3\) According to ISO/OSI, an SDU (Service Data Unit) is a packet or frame from the (n+1)-layer and treated like application data in the n-layer

\(^4\) According to ISO/OSI, a PDU (Protocol Data Unit) is a packet or frame on the n-layer
For each of the vehicle domains, i.e. Body-Electronics, Powertrain, Chassis and Telematic/-Infotainment, a middleware is defined. This middleware consists mainly of domain specific services. In this paper, the domain specific implementation of the body-electronics middleware is used as an example. An overview of the EAST-middleware for all domains can be found in [2]. Typically for the Body-Electronics domain is the high number of interconnected ECUs. Currently, the application software is considered as one piece of software while, as a rule, the operating system and the communication layer including network & power-management is realized by standard embedded software components. In EAST-EEA, this approach is extended for application software components too in order to show:

- Hardware independency: Show the feasibility of different suppliers for application software and ECU hardware.
- Portability: Software functions of a supplier can be deployed on ECUs provided by a different supplier.
- Interoperability: The EAST-EEA middleware provides common services used by application software components.

The corresponding ECU architecture is depicted in Figure 4. Each of the embedded software components provides dedicated services which can be accessed merely by means of the corresponding middleware interfaces. Besides an OSEK-operating system, the embedded basic software components for the Body Electronic ECU architecture are:

- A Hardware Abstraction Layer (HAL) used to encapsulate the micro-controller’s peripherals.
- An Actuator and Sensor Abstraction Layer (ASAL) to encapsulate the electrical & mechanical properties of sensors & actuators. An ASAL relates to a Local Device Manager (LDM) as defined in the AEE project.
- A Communication Abstraction Layer (CAL) encapsulation of communication services, i.e. protocols & physical links.
- The application is the collection of all application software components.
Communication between application software components is always done via the middleware, independently whether the application software components reside on the same ECU or on different ECUs. In Figure 5 application software component K “talks” to application software component L locally on ECU A, i.e. an application signal is transmitted. In fact, this is done by writing to an ECU-global variable via a standardized API. Depending on the implementation, this global variable resides in the middleware building block (solid connection) or the CAL building block (dashed connection). From the ADL point of view, this ECU-global variable is called in both cases an IPC-buffer. However, if application software component K talks to application software component N residing on ECU B, the ECU global variable used to exchange the application signal between the application software component and the CAN-driver resides in the CAL. The CAN-driver is part of the HAL – or COM-HAL to be precise- and will be accessed via the COM-HAL API from the CAL.

---

5 Steuergerät is the German word for ECU.
Sensor/Actuator access is performed by similar means. An ASAL component has a “hybrid” interface because it communicates via the middleware with application software components and via the HAL API with the micro-controller’s peripheral. The ASAL however is an application software component because it reflects all electrical & mechanical properties of the Sensor/Actuator. It is micro-controller independent because it transfers all actions via the μC specific HAL. In Figure 6, application software component K talks via the middleware with the ASAL for sensor S1 (solid line). A remote application software component K’ will talk to Sensor S1 by only using the appropriate middleware interface. Conceptually, the basic software components are invoked in the sequence CAL on ECU A, COM-HAL on ECU A, COM-HAL on ECU B, CAL on ECU B. CAL on ECU B will transfer the signal to the ASAL component via the middleware. Technically, this sequence is reflected by the OS- and bus-schedule whereas for each application signal there is an appropriate IPC buffer allocated at the ECUs and a slot in a communication systems’s frame. The allocation of application signals to IPC-buffers & communication systems frames is done statically at compile time.

Incorporating EAST Concepts

Figure 7 shows a typical design flow incorporating the EAST concepts of the ADL and the middleware. Starting with a description of the electronic features on vehicle level the application software is designed and analyzed independently from the hardware by means of the EAST-ADL. The behavior of elementary software functions is either described by using a native notation or external modeling tools. Application software components are obtained by applying code generation to logical clusters. Afterwards, they are mapped to ECUs according to non-functional requirements, e.g. cost. All basic software components are configured according to this mapping thus not introducing superfluous variables wasting memory. Configuration is applied to the OS (number & type of tasks), CAL & COM-HAL (mapping inter-ECU application signals to Frames), Middleware (IPC-Exchangers for Intra-ECU application signals), and ASALs (setting IPC-Exchangers & HAL API). The result is the operational architecture.

Conclusion

The EAST-EEA approach to manage the complexity uses a domain specific middleware for information exchange between application software components. These components are modeled by the Architecture description language.

---

summary 2nd European Congress ERTS
previous - 10 -
next 21 - 22 - 23 January 2004
Using a prototypical tool-environment, the EAST-ADL is currently being validated by the European automotive industry. Having integrated the validator feedback by mid 2004, a sound specification will be available to start commercial tool development just afterwards.

References